J Comput Virol (2011) 7:247–258
DOI 10.1007/s11416-011-0152-x

ORIGINAL PAPER

# Graph-based malware detection using dynamic analysis

**Blake Anderson · Daniel Quist · Joshua Neil · Curtis Storlie · Terran Lane**

Received: 7 March 2011 / Accepted: 20 May 2011 / Published online: 8 June 2011


**Abstract** We introduce a novel malware detection algorithm based on the analysis of graphs constructed from dynamically collected instruction traces of the target executable. These graphs represent Markov chains, where the vertices are the instructions and the transition probabilities are estimated by the data contained in the trace. We use a combination of graph kernels to create a similarity matrix between the instruction trace graphs. The resulting graph kernel measures similarity between graphs on both local and global levels. Finally, the similarity matrix is sent to a support vector machine to perform classification. Our method is particularly appealing because we do not base our classifications on the raw *n*-gram data, but rather use our data representation to perform classification in graph space. We demonstrate the performance of our algorithm on two classification problems: benign software versus malware, and the Netbull virus with different packers versus other classes of viruses. Our results show a statistically significant improvement over signature-based and other machine learning-based detection methods.

B. Anderson (✉) · D. Quist · J. Neil · C. Storlie
Los Alamos National Lab, Los Alamos, USA
e-mail: banderson@lanl.gov

D. Quist
e-mail: dquist@lanl.gov

J. Neil
e-mail: jneil@lanl.gov

C. Storlie
e-mail: storlie@lanl.gov

T. Lane
The University of New Mexico, Albuquerque, USA
e-mail: terran@cs.unm.edu

## 1 Introduction

Malware continues to be an ongoing threat to modern computing. It was recently estimated that one in four computers operating in the US are infected with malware [24]. In 2009, an estimated 55,000 malware samples were created, more than had appeared in the entire history of the computer virus up to that point [25]. With the ever increasing proliferation of these threats, it is important to develop new techniques to detect and contain these malware.

Many of the current antivirus programs available rely on a signature-based approach to classify programs as being either malicious or benign. Signature-based approaches are popular due to their low false positive rate and low computational complexity on the end host, both of which are appealing for daily usage. Unfortunately, these schemes have been shown to be easily defeated by simple code obfuscation techniques [9]. With the ease of creating a new virus through these techniques and polymorphic viruses becoming more prevalent, non-signature based methods are becoming more attractive.

To combat these issues, several researchers began to look at less strict measures to detect malicious code. These methods have generally revolved around *n*-gram analysis of the static binary or dynamic trace of the malicious program [11,27,28,36]. These methods have shown great promise in detecting zero-day malware, but there are drawbacks related to these approaches. The two parameters generally associated with *n*-gram models are $n$, the length of the subsequences being analyzed, and $L$, the number of $n$-grams to analyze. For larger values of $n$ and $L$, there is a much more expressive feature space that should be able to discriminate between malware and benign software more easily. But with these larger values of $n$ and $L$, we run into the *curse of dimensionality*: the feature space becomes too large and we do not have enough data to sufficiently condition the model. With smaller values of $n$ and $L$, the feature space becomes too small and discriminatory power is lost.

For our research, we use a modified version of the Ether Malware Analysis framework [12] to perform the data collection. Ether is a set of extensions on top of the Xen virtual machine. Malware frequently uses self-protection measures to thwart debugging and analysis. Ether uses a tactic of zero modification to be able to track and analyze a running system. Zero modifications preserve the sterility of the infected system, and reduce the methods that malware authors can use to detect if their malware is being analyzed. Increasing the complexity of detection makes for a much more robust analysis system. We use these modifications to allow for deeper introspection of the API and import internals [26].

Our data representation gets away from the need to specify the appropriate $n$ and $L$. Instead we model the data as a Markov chain represented by a weighted, directed graph. The instructions of the program are represented as vertices, and the weights of the edges are the transition probabilities of the Markov chain, which are estimated using the program trace we collect.

The novel contribution we present in this paper is to construct a similarity, or kernel, matrix between the Markov chain graphs and use this matrix to perform classification. We use two distinct measures of similarity to construct our kernel matrix: a local measure comparing corresponding edges in each graph and a global measure which compares aspects of the graphs' topologies. This combination allows us to compare the instruction trace graphs using very different criteria in a unified framework. Once the kernel matrix is constructed, we use support vector machines to perform the classification.

Our primary purpose is to show that our method outperforms $n$-gram and signature-based methods on the malware versus benign classification problem. To examine this problem, we use a dataset with 1,615 samples of malware and 615 samples of benign software. Our secondary purpose is to show that our algorithm can correctly discriminate between instances of the Netbull virus and other families of viruses. For these results, we use 13 samples of the Netbull virus with different packers and a random subsample consisting of 97 samples of unrelated malware. This result helps to validate the use of our similarity measure in a malware phylogenetic setting, as it shows the power of our method in classifying different examples of viruses. For malware phylogenetics/clustering, the objective is to determine which previously known malware samples are most similar to a newly detected piece of malware. This information would help malware researchers to more quickly understand the virus and perform the appropriate response [38].

Our paper is organized as follows: Sect. 2 examines our data collection strategy. Section 3 illustrates how we get the instruction trace data into our graph format, how we construct the kernel matrices between these graphs, and finally, how we classify the instruction traces. In Sect. 4 we demonstrate our results, Sect. 5 discusses related work, Sect. 6 mentions our ideas for future research directions, and we conclude in Sect. 7.

## 2 Data collection

Our data collection technique uses the Ether analysis framework to extract data from a Windows XP system. We chose the Ether system as it guarantees some level of protection against hardware based virtual machine detection. The primary mechanisms of protection that need to be overcome are debugger and virtual machine detection, timing attacks, and host system modifications. Each of these violate the fundamental tenet that the analyzed system must not be altered in any manner.

There are three specific detection techniques that justify the use of the Ether analysis framework. The first is based on the presence of a debugger. This is usually executed by the attacker reading the debugging flag from the process execution block of the running program. The Windows API IsDebuggerPresent flag indicates whether or not a debugger is watching the execution. This simple technique is enough to detect many of the common instrumentation systems in use today [23]. The second detection is the Red Pill class of instructions. Red Pill is a system developed by Rutkowska that detects the presence of a dynamically translated virtual machine such as VMWare or Virtual PC. In each of these virtual machines, the SIDT, store interrupt descriptor table, instruction will have a value that differs from a virtualized system and real hardware. Timing attacks, implemented with the RDTSC (read time step counter instruction) provide the third protection to be avoided. These attacks measure the time before and after a series of instructions. The difference between these times gives the attacker a very useful tool for determining if any monitoring is taking place (Fig. 1).

Any other modifications made to the analysis system must not be easily discoverable. Specifically, the primary method to be avoided is the "sterility" of the infected system. If there is any difference between a typical analysis system and a

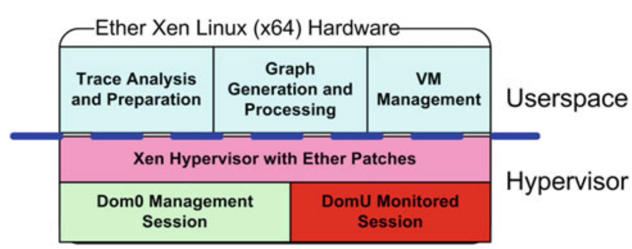



**Fig. 1** The architectural layout of the Ether integration

normal Windows system, this can be detected by the malware author.

The Ether system implements an instruction tracing mechanism that allows us to track the runtime execution of any process on the instrumented system. To find a process of interest, Ether parses and keeps track of the internal Windows kernel process list. When the process to be traced is scheduled by the Windows operating system, Ether makes note of the contents of the CR3 register, a unique identification that corresponds to the current process's page directory entry. From here, Ether uses two main methods to track individual instruction executions. First, the trap flag is set in the EFLAG register. This will cause a single-step trap to be raised to the operating system. Ether intercepts this trap at the hypervisor level, clears the EFLAG single-step bit, then marks the memory page for the code region invalid. Marking the memory page as invalid causes another trap to be generated, which is also intercepted. The EFLAG register is then reset, and the page error is cleared. This creates a back-and-forth operation that allows for single-stepping. To avoid detection by the monitored process, instructions that access the EFLAGS register are intercepted.

The end result of instruction tracing is a list of the in-order executed instructions. This instruction list is just a simple list of addresses, and the corresponding instruction. These instructions provide the input to our algorithms. These instruction monitoring systems have been shown to be successful in a wide range of operations [26].

To analyze the data, we begin by copying the executable to the Ether analysis system. Then an instantiation of a Windows virtual machine is started, and upon successful boot, the file is copied. At this time, the Ether portion of Xen is invoked and the malware is started. The sample is allowed to run for five minutes, which has been shown to be a sufficient time for execution [26].

## 3 Classification method

In this section we describe how we use the dynamic trace data to perform classification. Our method has two novel components: transforming the trace data into a Markov chain representation and using the graph kernel machinery to construct a similarity matrix between instances.

### 3.1 Data representation

Given an instruction trace $\mathcal{P}$, we are interested in finding a new representation, $\mathcal{P}'$, such that we can make unified comparisons in graph space while still capturing the sequential nature of the data. We achieved this by transforming the dynamic trace data into a Markov chain which we represent as a weighted, directed graph. A graph, $G = \langle V, E \rangle$, is composed of two sets, $V$ and $E$. The elements of $V$ are called vertices and the elements of $E$ are called edges. In our representation, the edge weight, $e_{ij}$, between vertices $i$ and $j$ corresponds to the transition probability from state $i$ to state $j$ in a Markov chain, hence, we require the edge weights for edges originating at $v_i$ to sum to 1, $\sum_{i \rightsquigarrow j} e_{ij} = 1$. We use an $n \times n$ ($n = |V|$) adjacency matrix to represent the graph, where each entry in the matrix, $a_{ij} = e_{ij}$.

| | |
|---|---|
| call | [ebp+0x8] |
| push | 0x70 |
| push | 0x010012F8 |
| call | 0x01006170 |
| push | 0x010061C0 |
| mov | eax, fs:[0x00000000] |
| push | eax |
| mov | fs:[], esp |
| mov | eax, [esp+0x10] |
| mov | [esp+0x10], ebp |
| lea | ebp, [esp+0x10] |
| sub | esp, eax |
| ... | ... |

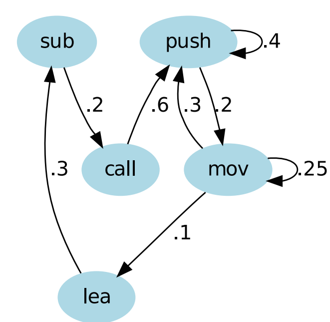



**Fig. 2** The *left* table shows an example of the trace data we collect. A hypothetical resulting graph representing a fragment of the Markov chain is shown on the *right*. In a real Markov chain graph, all of the out-going edges would sum to 1

We found 160 unique instructions across all of the traces we collected. These instructions are the vertices of the Markov chains. The 160 instructions are irrespective of the operands used with those instructions. By ignoring operands, we remove sensitivity to register allocation and other compiler artifacts. It is important to note that rarely did the instruction traces make use of all 160 unique instructions, and therefore, the adjacency matrices of the instruction trace graphs contain some rows of zeros. The decision to incorporate unused instructions in the model allowed us to maintain a consistent vertex set between all instruction trace graphs, granting us the ability to make uniform comparisons in graph space.

To find the edges of the graph, we first scan the instruction trace, keeping counts for each pair of successive instructions. After filling in the adjacency matrix with these values, we normalize the matrix such that all of the non-zero rows sum to one. This process of estimating the transition probabilities ensures us a well-formed Markov chain. Figure 2 shows a snippet of trace data with a resulting fragment of a hypothetical instruction trace graph. Our Markov chain graph can be summarized as $G = \langle V, E \rangle$, where

- $V$ is the vertex set composed of the 160 unique instructions,
- $E$ is the edge set where the transition probabilities are estimated from the data.

The graphs we construct approximate the pathways of execution of the program, and by using graph kernels (Sect. 3.2), we are able to exploit the local and global structure of these pathways. Also, unlike $n$-gram methods where we must choose the top-$Ln$-grams to use, doing our comparisons in graph space allows us to make implicit use of all the information contained in the instruction trace.

We experimented with another method for creating the instruction trace graphs, using a more expressive vertex set. In this method, we did not discard the arguments to the instructions but rather constructed vertices in the form $\langle operator, operand, operand \rangle$ where the operator is the instruction, and the operands are either null, or one of three types: register, memory, or dereference. This resulted in graphs with vertex sets of roughly 3,000 instructions. We did not use this representation due to poor initial performance with respect to accuracy and speed. We suspect this performance is a result of there not being enough trace data to accurately estimate the transition probabilities.

### 3.2 Constructing the similarity matrix

To make meaningful comparisons between the instruction trace graphs, we employed the techniques of graph kernels [18]. A kernel, $K(\mathbf{x}, \mathbf{x}')$, is a generalized inner product and can be thought of as a measure of similarity between two objects [31]. The power of kernels lies in their ability to compute the inner product between two objects in a possibly much higher dimensional feature space, without explicitly constructing the feature space. A kernel, $K : X \times X \to \mathbb{R}$, is defined as:

$$K(\mathbf{x}, \mathbf{x}') = \langle \phi(\mathbf{x}), \phi(\mathbf{x}') \rangle \tag{1}$$

where $\langle \cdot, \cdot \rangle$ is the dot product and $\phi(\cdot)$ is the projection of the input object into feature space. A well-defined kernel must satisfy two properties: it must be symmetric (for all $\mathbf{x}$ and $\mathbf{y} \in X$: $K(\mathbf{x}, \mathbf{y}) = K(\mathbf{y}, \mathbf{x})$) and positive-semidefinite (for any $x_1, \ldots, x_n \in X$ and $\mathbf{c} \in \mathbb{R}^n$: $\sum_{i=1}^{n} \sum_{j=1}^{n} c_i c_j K(x_i, x_j) \geq 0$). Kernels are appealing in a classification setting due to the *kernel trick* [31], which replaces inner products with kernel evaluations. The kernel trick uses the kernel function to perform a non-linear projection of the data into a higher dimensional space, where linear classification in this higher dimensional space is equivalent to non-linear classification in the original input space.

Our approach makes use of two types of kernels: a Gaussian kernel and a spectral kernel. The notions of similarity that these two kernels measure are quite distinct, and we found them to complement each other very well. The Gaussian kernel we use is:

$$K_G(\mathbf{x}, \mathbf{x}') = \sigma^2 e^{-\frac{1}{2\lambda^2} \sum_{i,j} (\mathbf{x}_{ij} - \mathbf{x}'_{ij})^2} \tag{2}$$

where $\mathbf{x}$ and $\mathbf{x}'$ are the weighted adjacency matrices of the Markov chains, $\sigma$ and $\lambda$ are the hyperparameters of the kernel function (estimated through cross-validation), and $\sum_{i,j}$ sums the squared distance between corresponding edges in the weighted adjacency matrices. This kernel searches for local similarities between the adjacency matrices. The motivation behind this kernel is that two different classes of programs should have different pathways of execution, which would result in a low similarity score.

The other kernel we use is based on spectral techniques [10]. These methods use the eigenvectors of the graph Laplacian to infer global properties about the graph. The weighted graph Laplacian is a $|V| \times |V|$ matrix defined as:

$$\mathcal{L} = \begin{cases} 1 - \frac{e_{vv}}{d_v} & \text{if } u = v, \text{ and } d_v \neq 0, \\ -\frac{e_{uv}}{\sqrt{d_u d_v}} & \text{if } u \text{ and } v \text{ are adjacent}, \\ 0 & \text{otherwise}. \end{cases} \tag{3}$$

where $e_{uv}$ is the weight between vertices $u$ and $v$, and $d_v$ is the degree of $v$. We take the eigenvectors associated with non-zero eigenvalues of $\mathcal{L}$, $\phi(\mathcal{L})$, as our new set of features. These eigenvectors encode global information about the graph's smoothness, diameter, number of components and stationary distribution among other things. With this information, we construct our second kernel by using a Gaussian kernel on the eigenvectors:

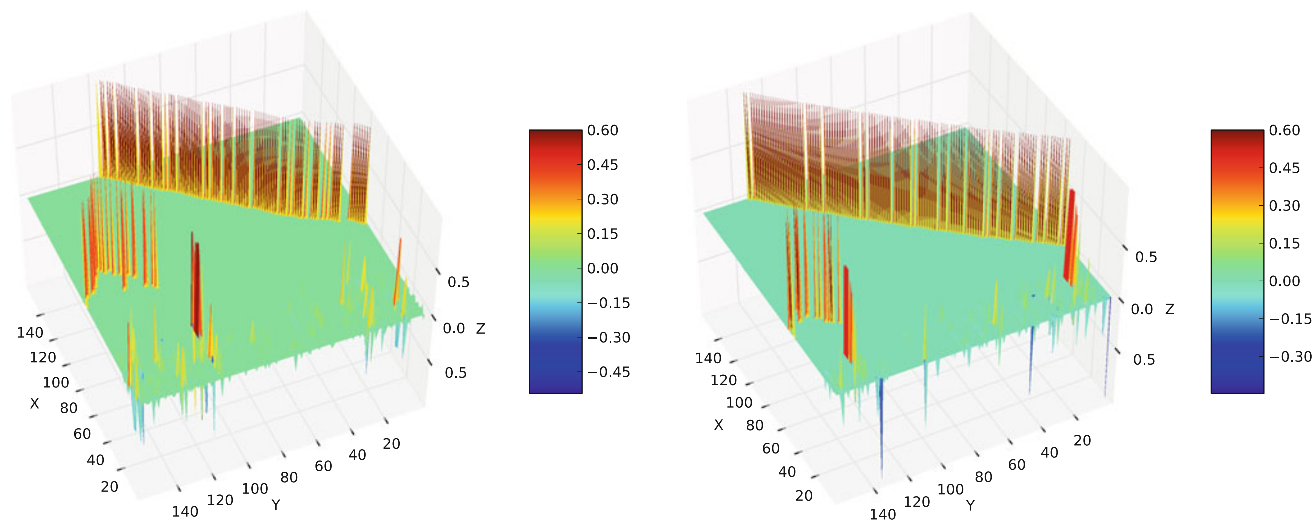
$$K_S(\mathbf{x}, \mathbf{x}') = \sigma^2 e^{-\frac{1}{2\lambda^2} \sum_k (\phi_k(\mathcal{L}(\mathbf{x})) - \phi_k(\mathcal{L}(\mathbf{x}')))^2} \tag{4}$$

where $\phi_k(\mathcal{L}(\mathbf{x}))$ and $\phi_k(\mathcal{L}(\mathbf{x}'))$ are the eigenvectors associated with weighted Laplacian of the adjacency matrices, $\mathcal{L}(\mathbf{x})$ and $\mathcal{L}(\mathbf{x}')$.

To give some intuition behind the spectral kernel, Figure 3 plots the eigenvectors of the graph Laplacian for an example of benign software and an example of malware. The diagonal ridge in the figure represents all of the unused instructions in the trace, which are disconnected components in the graph. To construct $K_S$ we only use the top-$k$ eigenvectors and this ridge information is discarded. The decision to use only the top-$k$ eigenvectors is defended in Sect. 4.3. The interesting information of the graph, the actual program flow contained in the largest connected component, is found in the spikes and valleys at the bottom of Fig. 3a, b. The eigenvectors of the Laplacian can be thought of as a Fourier basis for the graph [10]. Comparing these harmonic oscillations, encoded by the eigenvectors, between different types of software provides discrimination between structural features of the graph such as strongly connected components and cycles.

If we have two valid kernels, $K_1$ and $K_2$, we are assured that $K = K_1 + K_2$ is also a valid kernel [4]. This algebra on kernels allows us to elegantly combine kernels that measure very different aspects of the input data, and is the object of study in multiple kernel learning [2,35]. Our final kernel is a weighted combination of $K_G$ and $K_S$:

$$K_C = \mu K_G + (1 - \mu) K_S \tag{5}$$




**(a)** **(b)**

**Fig. 3** The eigenstructure of the Markov chain graph from two program traces. In **a** we have an example of benign software and in **b** we have an example of malware

where $0 \leq \mu \leq 1$. $\mu$ is found using a cross-validation search where we restrict candidate $\mu$'s to be in the range $[0.05, 0.95]$ with a step size of 0.05. Although more advanced techniques to search for the parameters of multiple kernel learning exist [35], we found this simple approach to be sufficient for the combination of these two kernels.

### 3.3 Classifying malware

We use support vector machines to perform the classification due to their intimate relationship with the kernels we construct. Support vector machines search for a hyperplane in the feature space that separates the points of the two classes with a maximal margin [6]. The hyperplane that is found by the SVM is a linear combination of our data instances, $x_i$, with weights, $\alpha_i$. It is important to note that only points close to the hyperplane will have non-zero $\alpha$'s. These points are called support vectors. Therefore, the goal in learning SVMs is to find the weight vector, $\alpha$, describing each data instance's contribution to the hyperplane. Using quadratic programming, we have the following optimization problem:

$$\max_{\alpha} \left( \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \langle \mathbf{x}_i, \mathbf{x}_j \rangle \right) \tag{6}$$

subject to the constraints:

$$\sum_{i=1}^{n} \alpha_i y_i = 0 \tag{7}$$

$$0 \leq \alpha_i \leq C \tag{8}$$

In Eq. 6, $y_i$ is the class label of instance $x_i$, and $\langle \cdot, \cdot \rangle$ is the Euclidean dot product. Equation 7 constrains the hyperplane to go through the origin. Equation 8 constrains the $\alpha$'s to be non-negative and less than some constant $C$. $C$ allows for soft-margins, meaning that some of the examples may fall between the margins. This helps to prevent over-fitting the training data and allows for better generalization accuracy. The weight vector for the hyperplane is then defined to be:

$$\mathbf{w} = \sum_{i} \alpha_i y_i \mathbf{x_i} \tag{9}$$

With this current setup, we are only afforded linear hyperplanes in the $d$-dimensional space defined by the feature vectors of $\mathbf{x}$. By using the *kernel trick*, we can project the data instances into a higher dimensional space and find a linear hyperplane in that space, which would be equivalent to a non-linear hyperplane in the original $d$-dimensional space. Our new optimization problem is:

$$\max_{\alpha} \left( \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j k(\mathbf{x}_i, \mathbf{x}_j) \right) \tag{10}$$

Equations 6 and 10 are identical with the exception that we have replaced the dot product, $\langle \cdot, \cdot \rangle$, with the kernel function $k(\cdot, \cdot)$.

Given $\alpha$ found in Eq. 10, we have the following decision function:

$$f(\mathbf{x}) = \operatorname{sgn} \left( \sum_{i}^{n} \alpha_i y_i k(\mathbf{x}, \mathbf{x}_i) \right) \tag{11}$$

which returns class +1 if the summation is $\geq 0$, and class $-1$ if the summation is $< 0$. The number of kernel computations in Eq. 11 is decreased because many of the $\alpha$'s are zero.

We implemented the computation of the kernel matrices ourselves and use the openly available PyML library [3] to perform the support vector machine training and classification. Reusing an openly available SVM implementation improves the reproducability of our results. The free parameter $C$ in Eq. 8 was estimated through cross-validation with the candidate values being [0.1, 1.0, 10.0, 100.0, 1,000.0].

### 3.4 Limitations

The computational complexity of this current method would be limiting in a real-time setting. It is important to note that the main kernel computation and the support vector machine optimization to find $\alpha$ could all be done offline and supplied to users. The kernel computations in Eq. 11 are $\mathcal{O}(n^2)$ for the Gaussian kernel and $\mathcal{O}(n^3)$ for the spectral kernel and would have to be done online. Although the sparsity of $\alpha$ helps to restrict the number of kernel computations required, this could still become expensive to the user. In Sect. 4.6 we present runtime results for our method and Sect. 6 discusses ways to speed up these operations.

## 4 Results

### 4.1 Data/environment

To perform our experiments, we used a machine with quad Xeon X5570s running at 2.93 GHz with 24 GB of memory. The dataset we used is composed of two distinct groups which we use to answer two main questions:

1. Can we correctly classify malicious software and benign software with a low false positive rate?
2. Can we differentiate the Netbull virus from other examples of malware?

To answer the first question, we collected 1,615 instances of malware and 615 instances of benign software, as described in Sect. 2. To answer the second question, we used 13 instances of the Netbull virus with different packers, such as UPX [13] and ASprotect [1], and compared these examples against a random subsample of 97 instances of malware. Using 13 different, commonly used packers provides extremely polymorphic versions of the same Netbull virus.

### 4.2 Other methods

To determine the validity of our method, we compared it against a standard $n$-gram model [19] and 9 leading antivirus software programs that use signature-based methods. For the standard $n$-gram model, we chose the top-$Ln$-grams to use by computing the information gain as suggested in [19]:

$$IG(j) = \sum_{v_j \in \{0,1\}} \sum_{y_i \in Y} P(v_j, y_i) \log \frac{P(v_j, y_i)}{P(v_j)P(y_i)} \tag{12}$$

where $v_j$ tells us whether the $j$th $n$-gram exists or not, $P(v_j, y_i)$ is the percentage of data instances of class $y_i$ with value $v_j$, $P(v_j)$ is the percentage of instances with value $v_j$, and $P(y_i)$ is the percentage of instances with class label $y_i$. Once we have sorted the list of $n$-grams based on their information gain, Eq. 12, we then select the top-$L$ to train the classification algorithm. It is important to note that we used $n$-grams derived from the same dynamic traces that our Markov chain representations were based on and not on static information contained in the binary.

To find the best choices for the parameters $n$ and $L$ for the standard $n$-gram model, we varied $n$ from 2 to 6 and $L$ from 500 to 3,000 in increments of 500. For each choice of parameters, we ran both a support vector machine, with a linear kernel, Gaussian kernel, and $d$-order polynomial ($2 \leq d \leq 9$) kernel, and a $k$-nearest neighbor classifier ($1 \leq k \leq 9$), with the feature vector consisting of the top-$Ln$-grams. We present the top-5 performing parameter combinations for the $n$-gram model and the top-5 performing antivirus programs in our results.

### 4.3 Selecting the $k$ eigenvectors

To find the appropriate $k$, the number of eigenvectors we use to classify the program traces in Eq. 4, we performed a series of tests on an independent dataset of 50 malware program traces and 10 benign traces where we adjusted $k$ using values ranging from 1 to 30. To ease computation, we would like to choose the smallest possible $k$ which still maintains discriminatory power. Using a multiple kernel learning framework allows us some freedom in choosing this parameter as the kernels work together to smooth each other. However, we did find that a near-optimal $k$ still has a statistically significant impact on the classification results.

Figure 4 shows the results of choosing $k$ averaged over 10 runs with the error bars showing one standard deviation. The decreasing performance as we increase $k$ is expected because as we choose more eigenvectors the feature space begins to overfit the training data, which results in lower accuracy on the test set. Using these results as a prior, we decided to set $k = 9$ for the other experiments we ran. Alternatively, $k$ could be selected using cross-validation on a validation dataset for each experiment.

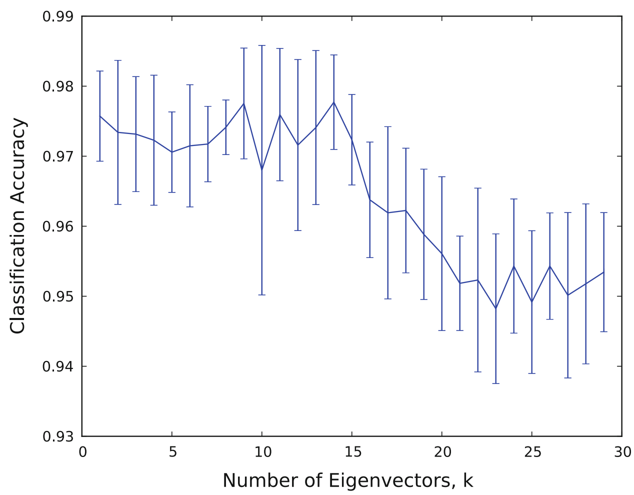



**Fig. 4** Classification accuracy of 50 instances of malware versus 10 instances of benign software as we vary the number of eigenvectors, $k$, of the spectral kernel. Results are averaged over 10 runs with the error bars being one standard deviation

### 4.4 Benign versus malware

We now explore the validity of our multiple kernel learning method as an alternative to $n$-gram and signature-based virus detection methods. Table 1 presents the results of our three different kernels and the $n$-gram methods using 10-fold cross-validation. The top-5 performing antivirus programs are also presented. For the $n$-gram methods, we used the same parameters as discussed in the previous section. The best results for the $n$-grams were achieved when $n = 4$, $L = 1{,}000$ and a support vector machine with a second order polynomial kernel was used. It is interesting to note that three antivirus programs, as well as our kernel methods, labeled the same benign executable as being malicious. This could be a result of a noisy dataset or a bad signature.

It is important to note that both machine learning approaches, graph kernels and $n$-grams, were able to easily out-perform the standard antivirus programs. Although $n$-grams were able to out-perform the antivirus programs, our results reinforce our hypothesis that learning with the Markov chain graphs improves accuracy over $n$-gram methods. Table 1 also illustrates that a combined kernel, which uses local and global structural information about the Markov chain graph, improves performance over the standalone kernels.

Figure 5 shows the heat maps (the values for the similarity matrix) for the 3 kernels we tested against. For visual purposes, we only show kernel values for 19 benign samples and 97 malware samples. The program traces that are more similar will have warmer colors. The block structure we see in this figure indicates that these kernels are able to discriminate between the two classes of software.

With our current dataset, we have more examples of malware than we do benign software. This is a by-product of the availability of the benign trace data. This data skew can in part be responsible for a portion of the false-positives we found in both our method and the $n$-gram methods. In a production setting, we would need a more diverse and extensive set of benign trace data in order to alleviate this problem.

### 4.5 Netbull versus malware

Our second set of experiments evaluates the performance of these algorithms with respect to their ability to differentiate between different types of malware. This is an important direction to pursue if we want to transfer this methodology to a clustering/phylogenetics setting. Our dataset was composed of 13 instances of the Netbull virus with different packers and a random subsample of 97 instances of malicious code from our main malware dataset. We limited the number

**Table 1** The classification accuracy of 615 instances of benign software versus 1,615 instances of malware

| Method | Accuracy (%) | FPs | FNs | AUC |
|---|---|---|---|---|
| Gaussian kernel | 95.70 | 44 | 52 | 0.9845 |
| Spectral kernel | 90.99 | 80 | 121 | 0.9524 |
| Combined kernel | **96.41** | 47 | **33** | **0.9874** |
| $n$-gram ($n = 3$, $L = 2{,}500$, SVM = 3-poly) | 82.15 | 300 | 98 | 0.9212 |
| $n$-gram ($n = 4$, $L = 2{,}000$, SVM = 3-poly) | 81.17 | 327 | 93 | 0.9018 |
| $n$-gram ($n = 2$, $L = 1{,}000$, 4-NN) | 80.63 | 325 | 107 | 0.8922 |
| $n$-gram ($n = 2$, $L = 1{,}500$, SVM = 2-poly) | 79.82 | 339 | 111 | 0.8889 |
| $n$-gram ($n = 4$, $L = 1{,}500$, SVM = Gauss) | 79.42 | 354 | 105 | 0.8991 |
| AV0 | 73.32 | **0** | 595 | N/A |
| AV1 | 53.86 | 1 | 1,028 | N/A |
| AV2 | 49.60 | **0** | 1,196 | N/A |
| AV3 | 43.27 | 1 | 1,264 | N/A |
| AV4 | 42.96 | 1 | 1,271 | N/A |

Statistically significant winners are bolded. The top 5 parameter choices for the $n$-gram model are presented as well as the top-5 performing signature-based antivirus programs

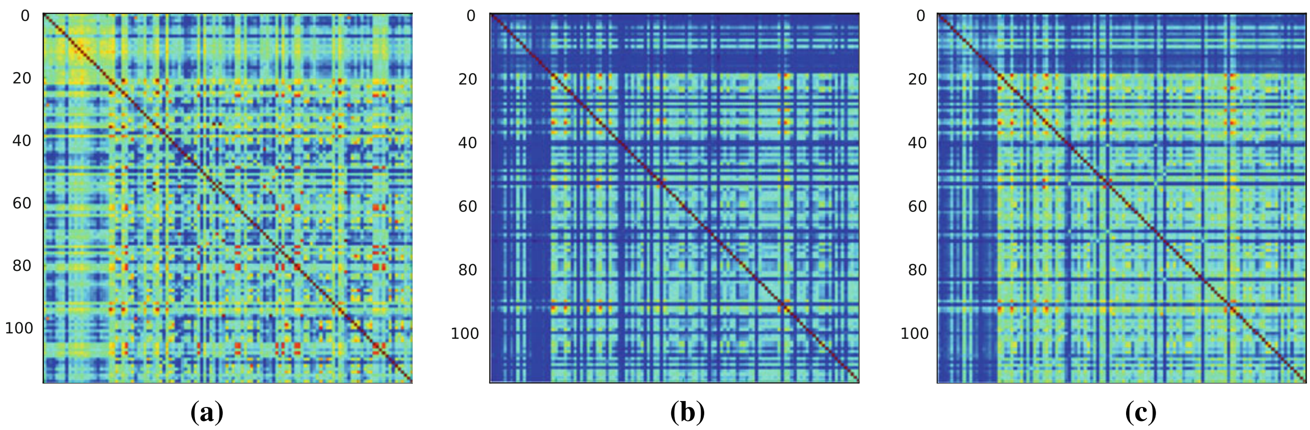



**Fig. 5** The heat maps of the kernel (similarity) matrix for benign software versus malware. The smaller block in the *upper left* of each figure is the benign software and the larger *lower right* is the malware. **a** Gaussian kernel, **b** Spectral kernel, **c** Combined kernel

**Table 2** The classification accuracy of 13 instances of the Netbull virus with different packers versus 97 instances of malware

| Method | Accuracy (%) | FPs | FNs | AUC |
|---|---|---|---|---|
| Gaussian kernel | **99.09** | **1** | **0** | **0.9965** |
| Spectral kernel | 96.36 | 4 | 0 | 0.9344 |
| Combined kernel | **100.00** | **0** | **0** | **1.00** |
| $n$-gram ($n = 4$, $L = 1{,}000$, SVM = 2-poly) | 94.55 | 5 | 1 | 0.8776 |
| $n$-gram ($n = 4$, $L = 2{,}500$, SVM = Gauss) | 93.64 | 6 | 1 | 0.8215 |
| $n$-gram ($n = 6$, $L = 2{,}500$, SVM = 2-poly) | 92.73 | 6 | 2 | 0.8432 |
| $n$-gram ($n = 3$, $L = 1{,}000$, SVM = 2-poly) | 89.09 | 12 | **0** | 0.6173 |
| $n$-gram ($n = 2$, $L = 500$, 3-NN) | 88.18 | 12 | 1 | 0.6334 |

Statistically significant winners are bolded

of other families of viruses to 97 due to the effects of data skew. The results are summarized in Table 2.

These results are very promising as our method using the combined kernel can correctly classify all instances of the Netbull virus despite this being a very skewed dataset. The $n$-gram methods had a more difficult time correctly classifying the instances of the Netbull virus given this extreme data skew. It is important to note that after the top-3 parameter choices for the $n$-grams, these models quickly devolved into predicting the majority class for all instances. This a common problem given data skew [7].

Our kernels for this dataset are displayed in Fig. 6 and have a similar block structure to Fig. 5. This is important as it validates our approach's ability to distinguish between somewhat similar pieces of malware. These results also validate using our data representation and associated kernels in a kernel-based clustering environment [22].

## 4.6 Timing results

In this section we explore the computation time for our method. As stated previously, there are two main components to our approach; computing the graph kernels and performing the support vector machine optimization (Eqs. 5 and 10), which can be done offline, and the classification of a new instance (Eq. 11), which is done online. We used the dataset composed of 1,615 samples of malicious programs and 615 samples of benign programs.

As Table 3 illustrates, the majority of our method's time is spent computing the kernel matrices. It took 698.45 s to compute the full kernel matrices. This may seem problematic, but since this portion can be done offline once, it will not slow down a production system. The online component of classifying a new instance took 0.54 s as shown in Table 3. The majority of this time is spent in computing the kernel values between the new instance and the labeled training data as described in Eq. 11.

The number of kernel computations is decreased due to the support vector machine finding a sparse set of support vectors. The PyML implementation of the SVM we used typically found ∼350 support vectors. There are other forms of support vector machines [16] that search for sparser solutions, which would help to speed up this online component by reducing the number of support vectors thereby reducing the number of kernel computations.

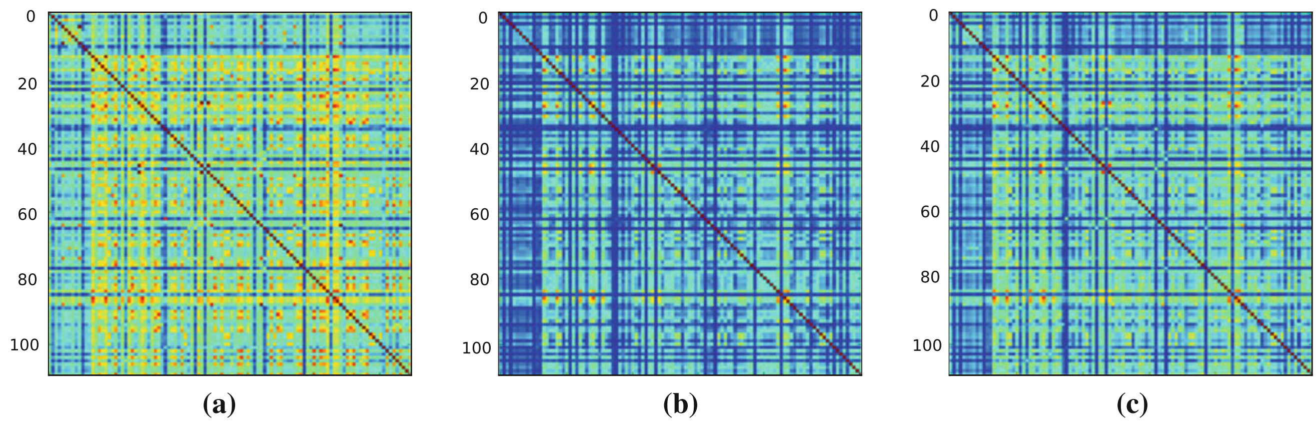



**Fig. 6** The heat maps of the kernel matrix for the Netbull virus with different packers versus malware. **a** Gaussian kernel, **b** Spectral kernel, **c** Combined kernel

**Table 3** Timing results for the computation time for each step of our method. All results are in seconds with one standard deviation given

| Component | Time |
|---|---|
| Gaussian kernel | 147.91 ± 9.54 |
| Spectral kernel | 550.55 ± 32.90 |
| SVM optimization | 0.16 ± 0.05 |
| Classifying new instance | 0.54 ± 0.07 |
| Total offline | 698.45 ± 57.44 |
| Total online | 0.54 ± 0.07 |

## 5 Related work

The $n$-gram methodology was one of the first malware detection techniques rooted in machine learning and has had great success at detecting obfuscated and polymorphic viruses [11,19,27,28]. Our method is related to $n$-grams in the sense that we use 2-grams to condition our transition probabilities for the Markov chain. The important distinction that our work makes is to treat this data in a novel way by using a graph multiple kernel learning framework.

The data transformation that we use, program trace to Markov chain, is similar to the Markov $n$-gram approach [32]. Here, they perform their analysis on static files such as MP3s, executable files, and compressed ZIP files and try to find an appropriate distribution of entropy for non-infected files. They set up a first order Markov chain between the bytes in their file, and compute the entropy using:

$$R = \sum_{i=0}^{n} \pi_i H(\mathbf{x}_i) \tag{13}$$

where $H(\mathbf{x}_i)$ is the entropy of row $i$ in the transition probability matrix. They then generate a histogram from this information and normalize it to obtain a distribution, which approaches the Gaussian distribution as the number of sampled entropies increases. They then use a threshold of 5 standard deviations from the mean to classify files as containing malware. In our work, we are not concerned with finding embedded malware but rather in classifying dynamic program traces. They also use the Markov chain in an information-theoretic setting whereas we use the graph structure present in the Markov chain to classify new traces.

Another line of research uses control flow graphs to categorize the behavior of the program [5,8,15,20]. The control flow graph is a representation that extracts all paths of execution that may be executed during a program's lifetime. In this approach, there is usually a code normalization piece that attempts to identify and remove unreachable code, simplify algebraic expressions, and get rid of intermediate variables [5]. With this preprocessing done, the control flow graph is constructed with the vertices being the basic blocks of the executable, and each edge is a possible flow of control between two vertices. The basic blocks are generally constructed using heuristics on the static disassembled executable based on jump, call, and return instructions. Classification based on control flow graphs generally uses sub-graph matching [5,20] or edit-distance between graphs [8].

Our method differs from these control flow graph methods in several respects. We use data derived from the dynamic execution of a program as opposed to analysis on the disassembled binary. We do not group this information into basic blocks, but instead use the Markov chain representation of individual instructions to arrive at a compact representation that grants us a finer level of resolution. To perform the classification, we use kernel methods that look at the global and local similarity between graphs, whereas the control flow graph methods use either sub-graph isomorphism, which there are no known general polynomial time algorithms and therefore must rely on heuristics, or edit-distance, which ignores the global structure of the graphs.

The closest work to ours analyzes the system calls performed by a program [37]. They use two representations in their analysis. The first is a tree structure where the vertices are either processes or system calls, and the edges are created when a process creates another process or makes a system call. The second is a tree structure where the vertices are processes and each vertex contains information about the probability distribution for the executed system calls. Edges are created when one process creates another. In contrast, we perform our analysis on the Markov chain representation of the instruction trace of a program, which has the same advantages compared to the control flow graphs: finer level of resolution and more expressive power. The kernel they use on the first data representation counts the number of common sub-trees contained in each tree. The kernel used on the second data representation works by enumerating random walks of differing lengths, and then comparing them with

$$K\left(\mathbf{x}, \mathbf{x}'\right) = K_v\left(v_1, v_1'\right) \prod_{i=2}^{l} K_e\left(e_{i-1:i}, e_{i-1:i}'\right) K_v\left(v_i, v_i'\right) \tag{14}$$

where $K_v(v_i, v_i')$ is the Gaussian kernel between the system call probability distributions at those two vertices, and $K_e(e_{i-1:i}, e_{i-1:i}')$ is assumed to be 1. The kernels we use do not have to enumerate the random paths of some length, $l$, as they infer the global structure of the graph using spectral graph techniques.

## 6 Future work

There are some significant limitations of the Ether system. First, Ether is not completely invisible. Recent variants of the RDG Tejon packer have been able to detect Ether [34]. Specifically, the BIOS data string for Ether uses an emulated variant from the Bochs virtual machine [21], and can be detected. Second, the Ethernet card that is emulated by the underlying Xen system can be easily analyzed. Each of these string settings can be changed to make sure that they are not detected. The third and final problem with Ether is that it is a slow system for analysis. Each instruction generates an interrupt in the form of a page fault or a debug exception. Generating these interrupts causes significant performance problems that are inherent in this analysis. Therefore, Ether is unsuitable as a system to be implemented in production or consumer Windows implementations.

Although the results we present show great promise in using the graph structure of the instruction traces to classify malware, our current method's computational complexity would be prohibitive in a real-time setting. The classification method is made up of two main components: an offline component that constructs the kernel matrix and finds the support vectors of the system, and an online component that classifies new program traces as being either malicious or benign. It is important to note that the runtime of the offline element is of less importance as this will not impact the user.

The worst-case computational complexity for solving the optimization problem, Eq. 10, is $\mathcal{O}(n^3)$ where $n$ is the number of support vectors [6]. Although this is done offline, there are several alternative SVM approaches, such as the reduced support vector machine [16], that would help to increase the speed of computing the weights of the support vectors.

The computationally intensive piece of the online component is evaluating the kernel in Eq. 11. Currently, we are naïvely computing the eigenvectors for Eq. 4 using a singular value decomposition. This operation is $\mathcal{O}(n^3)$ and is unnecessary as it computes all of the eigenvectors and we only use the top-$k$. Instead, we could use Hotelling's power method to find the top-$k$ eigenvectors [14], where $k \ll n$. This method runs in $\mathcal{O}(kn^2)$ and would help to increase the speed of both the offline complete kernel computation, and the online computations of Eq. 11.

The multiple kernel learning framework gives us a logical way to measure different aspects of the program trace data that we have collected. An interesting direction would be to incorporate different data sources, each with appropriate kernels, into our composite kernel. These data sources could include information based on the static analysis of the binary and the API sequence calls made by the program. Methods based on these data sources have been shown to be successful [30,33,39], and could possibly lead to more accurate results when combined in our multiple kernel learning framework.

In our current setup, we were able to naïvely learn $\mu$ in Eq. 5 through cross-validation. If we decide to add different types of kernels or use multiple data sources, this naïve approach will no longer be suitable. Instead, we can embed the multiple kernel learning within the support vector machine's optimization problem, now a semi-infinite linear program, which allows us to simultaneously find the support vectors and the new parameter $\beta$ [35]. $\beta$ is the parameter that controls the contribution of each kernel with the constraint $\sum_{i=0}^{k} \beta_i = 1$.

Recently, there has been a rising interest in learning how to cluster malware so that researchers can gain insight into the phylogenetic structure of current viruses [17,29]. Because the majority of new viruses are derived from, or are composites of, established viruses, this information would allow for more immediate responses and allow researchers to understand the new virus much more quickly. Given our kernel matrix, which we have shown can correctly classify the Netbull viruses against other types of viruses, we can use spectral clustering [22]. With spectral clustering, we aim to use the eigenstructure of the kernel matrix to cluster the different data instances into different families.

## 7 Conclusion

With the advent of polymorphic code and obfuscated viruses, signature-based malware detection is becoming quickly outdated [9]. To combat this, many researchers have began drawing ideas from machine learning to create more flexible detection algorithms. Many of these approaches have centered around using *n*-gram based statistics to classify new instances as being either benign or malware.

Our novel method extends the *n*-gram methodology by using 2-grams to condition the transition probabilities of a Markov chain, and then treats that Markov chain as a graph. Taking the Markov chain as a graph allows us to utilize the machinery of graph kernels to construct a similarity matrix between instances in our training set. We use two distinct measures of similarity to construct our kernel matrix: a Gaussian kernel, which measures local similarity between the graphs' edges, and a spectral kernel, which measures global similarity between the graphs. Given our kernel matrix, we can then train a support vector machine to perform classification on new testing points.

Using the hardware hypervisor affords us a unique look into the running program currently unavailable from more traditional debugger based methods. The lowered detectability, and the protections afforded to a Xen virtualized system, make this a compelling method for our data collection.

We have demonstrated the performance of our multiple kernel learning framework on three problems. The first problem investigated whether our method could properly discriminate between instances of malware and benign software. We showed that with our combined kernel we were able to outperform *n*-gram and signature-based methods, while maintaining a low false positive rate.

The second problem tested our method in its ability to discriminate between different types of malware. We compared the Netbull virus with different packers to a set of other instances of malware. The multiple kernel learning method was able to perfectly classify these instances. This result shows great promise for using these kernels in a clustering setting, which would allow researchers to more quickly understand the dynamics of a new virus.

**Acknowledgments** Dr. Lane's work was supported by the NSF under grant IIS-0705681. This material was prepared by Los Alamos National Security, LLC (LANS) under Contract DE-AC52-06NA25396 with the U.S. Department of Energy (DOE).

## References

1. Aspack software. http://www.aspack.com/asprotect.html, Accessed 5 August 2010
2. Bach, F.R., Lanckriet, G.R.G., Jordan, M.I.: Multiple kernel learning, conic duality, and the smo algorithm. In: Proceedings of the Twenty-First International Conference on Machine Learning, ICML'04, p. 6. ACM, New York (2004)
3. Ben-Hur, A.: Pyml: machine learning in python. http://pyml.sourceforge.net/, Accessed 28 July 2010
4. Bishop, C.M.: Pattern Recognition and Machine Learning (Information Science and Statistics). Springer, New York (2006)
5. Bruschi, D., Martignoni, L., Monga, M.: Detecting self-mutating malware using control-flow graph matching. In: Bschkes, R., Laskov, P. (eds.) Detection of Intrusions and Malware and Vulnerability Assessment. Lecture Notes in Computer Science, vol. 4064, pp. 129–143. Springer, Berlin (2006)
6. Burges, C.J.C.: A tutorial on support vector machines for pattern recognition. Data Min. Knowl. Discov. **2**, 121–167 (1998)
7. Cardie, C., Nowe, N.: Improving minority class prediction using case-specific feature weights. In: Proceedings of the Fourteenth International Conference on Machine Learning, ICML'97, pp. 57–65. Morgan Kaufmann Publishers Inc, San Francisco (1997)
8. Cesare, S., Xiang, Y.: Classification of malware using structured control flow. In: Proceedings of the Eighth Australasian Symposium on Parallel and Distributed Computing, vol. 107, AusPDC '10, pp. 61–70. Australian Computer Society Inc, Darlinghurst (2010)
9. Christodorescu, M., Jha, S.: Static analysis of executables to detect malicious patterns. In: In Proceedings of the 12th USENIX Security Symposium, pp. 169–186 (2003)
10. Chung, F.R.K.: Spectral Graph Theory (CBMS Regional Conference Series in Mathematics, No. 92). American Mathematical Society, Providence (1997)
11. Dai, J., Guha, R., Lee, J.: Efficient virus detection using dynamic instruction sequences. J. Comput. **4**(5), 405–414 (2009)
12. Dinaburg, A., Royal, P., Sharif, M., Lee, W.: Ether: malware analysis via hardware virtualization extensions. In: Proceedings of the 15th ACM conference on Computer and communications security, CCS '08, pp. 51–62. ACM, New York (2008)
13. UPX: The Ultimate Packer for eXecutables. http://upx.sourceforge.net/, Accessed 16 August 2010
14. Hotelling, H.: Analysis of a complex of statistical variables into principal components. J. Educ. Psychol. **24**(6), 417–441 (1933)
15. Hu, X., Chiueh, T.-c., Shin, K.G.: Large-scale malware indexing using function-call graphs. In: Proceedings of the 16th ACM Conference on Computer and Communications Security, CCS'09, pp. 611–620. ACM, New York (2009)
16. Lee, Y.J., Mangasarian, O.L.: Rsvm: reduced support vector machines. In: Data Mining Institute, Computer Sciences Department, University of Wisconsin, pp. 00–07 (2001)
17. Karim, Md, Walenstein, A., Lakhotia, A., Parida, L.: Malware phylogeny generation using permutations of code. J. Comput. Virol. **1**, 13–23 (2005)
18. Kashima, H., Tsuda, K., Inokuchi, A.: Kernels for Graphs. MIT Press, Massachusetts (2004)
19. Kolter, J.Z., Maloof, M.A.: Learning to detect malicious executables in the wild. In: KDD '04: Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 470–478. ACM, New York (2004)
20. Kruegel, C., Kirda, E., Mutz, D., Robertson, W., Vigna, G.: Polymorphic worm detection using structural information of executables. In: Valdes, A., Zamboni, D. (eds.) Recent Advances in Intrusion Detection. Lecture Notes in Computer Science, vol. 3858, pp. 207–226. Springer, Berlin (2006)
21. Lawton, K., Denney, B., Guarneri, N.D., Ruppert, V., Bothamy, C.: Bochs user manual. Online user manual, November 2010
22. Luxburg, U.: A tutorial on spectral clustering. Stat. Comput. **17**(4), 395–416 (2007)
23. Microsoft, Inc. IsDebuggerPresent function. http://msdn.microsoft.com/en-us/library/ms680345(VS.85).aspx, October 2010
24. Organisation for Economic Co-operation and Development. Malicious software (malware): A security threat to the internet economy. White Paper, June 2008

25. Panda Security. Panda labs annual report 2009. White Paper, January 2010
26. Quist, D., Liebrock, L., Neil, J.: Improving antivirus accuracy with hypervisor assisted analysis. J. Comput. Virol 1–11 (2010). doi:10.1007/s11416-010-0142-4
27. Reddy, D., Dash, S., Pujari, A.: New malicious code detection using variable length *n*-grams. In: Information Systems Security. Lecture Notes in Computer Science, vol. 4332, pp. 276–288. Springer, Berlin (2006)
28. Reddy, D., Pujari, A.: N-gram analysis for computer virus detection. J. Comput. Virol. **2**, 231–239 (2006)
29. Rieck, K., Holz, T., Willems, C., Dssel, P., Laskov, P.: Learning and classification of malware behavior. In: Zamboni, D. (ed) Detection of Intrusions and Malware, and Vulnerability Assessment. Lecture Notes in Computer Science, vol. 5137, pp. 108–125. Springer, Berlin (2008)
30. Wang, K., Stolflo, S.J., Li, W.J.: Fileprint analysis for malware detection. In: ACM CCS WORM (2005)
31. Schölkopf, B., Smola, A.J.: Learning with Kernels. MIT Press, Massachusetts (2002)
32. Shafiq, M., Khayam, S., Farooq, M.: Embedded malware detection using markov n-grams. In: Detection of Intrusions and Malware, and Vulnerability Assessment. Lecture Notes in Computer Science, vol. 5137, pp. 88–107. Springer, Berlin (2008)
33. Shankarapani, M., Ramamoorthy, S., Movva, R., Mukkamala, S.: Malware detection using assembly and api call sequences. J. Comput. Virol. pp. 1–13 (2010). doi:10.1007/s11416-010-0141-5
34. RDGMax Software. RDG Tejon Crypter. Software package, November 2010
35. Sonnenburg, S., Raetsch, G., Schaefer, C.: A general and efficient multiple kernel learning algorithm (2006)
36. Stolfo, S., Wang, K., Li, W.J.: Towards stealthy malware detection. In: Malware Detection. Advances in Information Security, vol. 27, pp. 231–249. Springer, Berlin (2007)
37. Wagner, C., Wagener, G., State, R., Engel, T.: Malware analysis with graph kernels and support vector machines. In: Malicious and Unwanted Software (MALWARE), 2009 4th International Conference, pp. 63–68 (2009)
38. Walenstein, A., Venable, M., Hayes, M., Thompson, C., Lakhotia, A.: Exploiting similarity between variants to defeat malware (2008)
39. Li, T., Ye, Y., Wang, D., Ye, D.: Imds: Intelligent malware detection system. In: Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (2007)